Object Name	OID	Access	Description
System Parameters 1.3.6.1.4.1.1000.1.1			
SysVersion	1	Read Only (RO)	The major and minor version number of the installed System software, including the version of the custom proxy program at the gateway server if applicable (so that a new SNMP extension agent is not also required at the gateway server).
SysMaxNoClients	2	*Read /Write (R/W)*	The number of hotel rooms connected to this System server.
System Intermediate Driver Parameters 1.3.6.1.4.1.1000.1.2			
IntIPAddr	1	R/W	The client-side network adapter's IP address (this is the IP address all destination IP addresses are changed into when an IP packet is received).
IntDestAddrPool	2	Not Accessi-ble (NA)	A table of IntDestAddrPoolEntry's. Each IntDestAddrPoolEntry shall contain the source IP (RO), the source port (RO), the destination IP (RO), the flags (RO), the source MAC address (RO) and the TTL value (R/W). The TTL value shall be used to remove entries from the table (i.e. by setting the value to 0).
IntMaxConn	3	R/W	The size of DestAddrPool, which determines the number of connections which can be supported over the server.
ARP Spoofer Parameters 1.3.6.1.4.1.1000.1.3			
ArpHardwareID	1	R/W	A string specifying the Ethernet card to listen on.
ArpHardwareAddr	2	R/W	The Ethernet address of the client-side network adapter of the System server.
DNS Spoofer-Forwarder Parameters 1.3.6.1.4.1.1000.1.4			
DNSRetIPAddr	1	R/W	The client-side network adapter's IP address. This value is returned in response to DNS queries intercepted by the System server.
DNSDomNameTable	2	NA	A table of DNSDomNameEntry's. Each DNSDomNameEntry shall contain the domain name (RO), destination IP (RO) and corresponding table of source IPs (RO).

Table 1

Trap Name	Enterprise ID	Generic Trap Number	Specific Trap Number	Variables	Description
DestAddrPool Full	1.3.6.1.4.1. 1000.1 (iso.org.dod.internet.private.enterprises.ficotional.readynet)	6 (always 6 for enterprise-specific traps)	1	IntMaxConn, SysMaxNoClients, unique trap ID (always generated for the Trap PDU header), time stamp (always collected for the Trap PDU header)	This trap is sent whenever DestAddrPool attempts to store n+1 entries, where n is equal to the maximum size of the table.

Table 2

Active Routes			
Network Destination	Netmask	Gateway	Interface
0.0.0.0	0.0.0.0	200.10.5.1	200.10.5.1
222.10.10.2	255.255.255.255	200.10.10.1	200.10.10.1
222.10.10.3	255.255.255.255	200.10.10.1	200.10.10.1
…	…	…	…
200.10.10.24	255.255.255.255	200.10.10.1	200.10.10.1
200.10.10.25	255.255.255.255	200.10.15.120	200.10.15.120

Table 3

APPENDIX

Module Main.cpp

Main.cpp_2(2)

FUNCTION
ControlHandler

```
BOOL WINAPI
ControlHandler(DWORD dwCtrlType)
{
  switch(dwCtrlType) {
    case CTRL_C_EVENT:
    case CTRL_BREAK_EVENT
    case CTRL_CLOSE_EVENT:
    case CTRL_LOGOFF_EVENT:
    case CTRL_SHUTDOWN_EVENT
      RnetLogger("MAIN", "User invoked shutdown", RNET_LOG_0);
      ReadyNetCleanUp();
      break;
  }
  return true;
}
```

FUNCTION
ReadConfigFile

```
bool
ReadConfigFile(void)
{
  FILE *ConfigFile = NULL;
  char parm[50], value[30], *p;
  int ParmsFound = 0;

  ConfigFile  = fopen(ConfigFileName, "r");

  if (ConfigFile != NULL) {

    while (!feof(ConfigFile)) {
      parm[0]=0;
      fscanf(ConfigFile, "%s %s", &parm, &value);
      /* Convert to upper case */
      for( p = parm; p < parm + strlen( parm ); p++ ){
        if( islower( *p ) )
          *p = _toupper( *p );
      }
      if ( strcmp(parm, "LOG_MESSAGE_LEVEL") == 0) {
        allowed_msg_level = atoi(value);
        ParmsFound++;
      }
      else if (strcmp(parm, "RNET_DNS_INET_ADDR") == 0) {
        strcpy(rnet_dns_inet_addr, value); ParmsFound++;
      }
      else if( strcmp(parm, "RNET_SERVER_INET_ADDR") == 0) {
        strcpy(rnet_server_inet_addr, value), ParmsFound++;
      }
      else if( strcmp(parm, "ADAPTERNAME") == 0) {
        strcpy(adapter_name_value, value); ParmsFound++;
      }
      else if (strcmp(parm, "MACADDRESS") == 0) {
        strcpy(mac_address_value, value), ParmsFound++;
      }
      else if (strcmp(parm, "SPLASHTYPE") == 0) {
        strcpy(splash_page_type, value), ParmsFound++;
      }
      else if (strcmp(parm, "") == 0) {
      }
    }

    fclose(ConfigFile);
    if (ParmsFound == 6)
      return true;
    else {
      RnetLogger("MAIN", "RNETCONF.TXT - Check number input parameters", RNET_ERROR_LOG);
      return false;
    }
  }
  else {
    RnetLogger("MAIN", "File RNETCONF.TXT does not exist", RNET_ERROR_LOG);
    return false;
  }
}
```

FUNCTION
getDestIp

```
int
getDstIp(int src_ip, short src_port)
{
 /*
 ** Get the entry from the client table
 */
 int c_index = getClientIdx(src_ip),
 if (c_index == NOT_FOUND)
  return FAILURE,

 /*
 ** Get the connection info from the clients connection table entry
 */
 conn_tbl_tp * conn_entry = getConnPtr(c_index, src_port),
 if (conn_entry == NULL)
  return FAILURE,

 return conn_entry->dst_ip,
}
```

FUNCTION
connectToHost

```
int
connectToHost(int ip, unsigned short port)
{
 struct sockaddr_in host_addr;

 /*
 ** Obtain a TCP socket.
 */
 bzero((char *) &host_addr, sizeof(host_addr));

 host_addr.sin_family      = AF_INET;
 host_addr.sin_addr.s_addr = ntohl(ip);
 host_addr.sin_port        = ntohs(port);

 int sock = w_socket(AF_INET, SOCK_STREAM, 0),

 if (sock == INVALID_SOCKET){
  RnetLogger("ERROR",
          "Could not obtain socket to connect to host",
          RNET_ERROR_LOG),
  return FAILURE;
 }

 /*
 ** Attempt to connect to the server.
 */
 int rc = w_connect(sock, (struct sockaddr *) &host_addr, sizeof(host_addr)),

 if (rc == SOCKET_ERROR) {
  RnetLogger("ERROR",
          "An error occurred trying to connect to host",
          RNET_ERROR_LOG);
  return FAILURE;
 }

 return sock,
}
```

FUNCTION
proxyMessage

```
int
proxyMessage(int src_sock, int dst_sock)
{
 char msg[MAX_MSG_SIZE],
 int len;

 /*
 ** Receive a message from the source socket.
 */
 if ((len = w_recv(src_sock, msg, MAX_MSG_SIZE, 0)) <= 0)
  return FAILURE,

 /*
 ** Proxy the message to the destination socket.
 */
 if (w_send(dst_sock, msg, len, 0) <= 0)
  return FAILURE;

 return SUCCESS;
}
```

FUNCTION
dataHandler

```
void dataHandler(void *param)
{
 SOCKET socketfd, cli_socketfd, serv_socketfd;
 int rc,
 fd_set fdvar_read;

 conn_info_tp2 *conn_info2 = (conn_info_tp2*) param,
 socketfd = conn_info2->server;
 cli_socketfd = conn_info2->client;

 serv_socketfd = w_accept(socketfd, NULL, NULL);
 if (serv_socketfd == INVALID_SOCKET)
          throw;

 for(;;){
          // set (or reset) the structure
          FD_ZERO(&fdvar_read),
          // Cast to unsigned int to get rid of compiler warning
          FD_SET((unsigned int) cli_socketfd, &fdvar_read),
          FD_SET((unsigned int) serv_socketfd, &fdvar_read),

          // Wait for activity on either socket.
          rc = w_select((int) NULL,
                                          &fdvar_read,
                                          (fd_set *) NULL,
                                          (fd_set *) NULL,
                                          (const struct timeval *) NULL);

          if (rc == SOCKET_ERROR) {
           // RNET_LOGGER("ERROR", "Select socket function returned an error",
            RNET_ERROR_LOG),
            break,
          }

          // Receive from client and send to server
          if (FD_ISSET(cli_socketfd, &fdvar_read) != 0) {
          rc = proxyMessage(cli_socketfd, serv_socketfd);
          if (rc != SUCCESS)
             break;
          }

          // Receive from server and send to client
          if (FD_ISSET(serv_socketfd, &fdvar_read) != 0) {
            rc = proxyMessage(serv_socketfd, cli_socketfd),
            if (rc != SUCCESS)
              break;
          }
 }
 w_closesocket(cli_socketfd),
 w_closesocket(serv_socketfd),
 delete(conn_info2),
}
```

FUNCTION
openDataConnection

```
void openDataConnection(char buffer[MAX_MSG_SIZE], void *param, int s_sock)
{
 char *firStr, *secStr, tempStr[16],
 int firNum, secNum, old_port_num, new_port_num,
 int i, rc,
 int cli_socketfd, serv_socketfd,
 struct sockaddr_in c_socket, s_socket;

 conn_info_tp *conn_info = (conn_info_tp*) param,

 // strtok the original client side port number
 strtok(buffer, " "),
 for(i=0,i<5,i++) firStr = strtok(NULL, ",");
 secStr = strtok(NULL, ","),
 firNum = atoi(firStr);
 secNum = atoi(secStr),
 old_port_num = firNum*256 +secNum,
 new_port_num = old_port_num,

 try {
        // initialize the client side socket
        cli_socketfd = w_socket(AF_INET, SOCK_STREAM, 0);
        bzero((char *) &c_socket, sizeof(c_socket)),

        c_socket.sin_family = AF_INET,
        c_socket.sin_addr s_addr = htonl(conn_info->src_ip),
        c_socket.sin_port = htons(old_port_num),

        if (cli_socketfd == INVALID_SOCKET) return;

        rc = w_connect(cli_socketfd, (struct sockaddr *) &c_socket, sizeof(c_socket)),

        if (rc == SOCKET_ERROR) return,

        // initialize the server side socker
        for (;;) {
          serv_socketfd = socket(AF_INET, SOCK_STREAM, 0);
          bzero((char *) &s_socket, sizeof(s_socket));
          s_socket.sin_family = AF_INET,
          s_socket.sin_addr.s_addr = inet_addr(met_server_inet_addr),
          s_socket.sin_port = htons(new_port_num);
          rc = w_bind(serv_socketfd, (struct sockaddr *) &s_socket, sizeof(s_socket)),

          if (rc == SOCKET_ERROR) throw("binding server socket"),

          rc = w_listen(serv_socketfd, BACKLOG_SIZE_FTP);
          /* if the port is already used, try the next higher port */
          if (rc == WSAEISCONN) {
            new_port_num = new_port_num + 1,
                    continue;
          }

          if (rc == SOCKET_ERROR) throw,

          // reformat the msg with new port new and send to the server.
          strcpy(tempStr, met_server_inet_addr),
          sprintf(buffer, "PORT %s", strtok(tempStr, ".")),
          for(i=0; i<3, i++)
          sprintf(buffer, "%s,%s", buffer, strtok(NULL, ".")),
          firNum = new_port_num % 256,
          secNum = (new_port_num - firNum)/256,
          sprintf(buffer, "%s,%d,%d\r\n",buffer, secNum, firNum);

          if (w_send(s_sock, buffer, strlen(buffer), 0) < 0)  return,

          conn_info_tp2 *conn_info2 = new conn_info_tp2;
          conn_info2->client = cli_socketfd,
          conn_info2->server = serv_socketfd,
          rc = _beginthread(dataHandler, 0, (void *) conn_info2);
          if (rc <0) throw;
          else break;
        } // for
 } //try
 catch (. .) {
        /* Who knows what happened  Just shutdown and restart the service */
        w_closesocket(serv_socketfd); w_closesocket(cli_socketfd),
//      RNET_LOGGER("FTP", msg, RNET_ERROR_LOG);
        ThreadMtceTable[FTP].inService = false,
        num_threads_running --,
        _endthread(),
 }
}
```

FUNCTION
proxyMessages

```
void
proxyMessages(void *param, int s_sock)
{
 int c_sock,
 char msg[MAX_MSG_SIZE],
 int len,

 conn_info_tp *conn_info = (conn_info_tp*) param,
 c_sock = conn_info->c_sock;

 for( , ; ) {
  /* Listen to both sockets for activity (a send or a receive ) */
  fd_set select_set,
  FD_ZERO(&select_set),
  FD_SET((unsigned int) c_sock, &select_set),
  FD_SET((unsigned int) s_sock, &select_set);
  int rc = w_select((int) NULL,
                &select_set,
                (fd_set *) NULL,
                (fd_set *) NULL,
                (const struct timeval *) NULL),

  if (rc == SOCKET_ERROR) { RnetLogger("ERROR","Select socket function returned an error",
          RNET_ERROR_LOG);
   return;
  }

  /* Proxy the message to the server if the client sent it */
  if (FD_ISSET(c_sock, &select_set) != 0) {
            if ((len = w_recv(c_sock, msg, MAX_MSG_SIZE, 0)) <= 0)
                        return,
            /* if FTP command PORT is found, open data connection */
            if (strstr(msg, "PORT") != NULL)
                        openDataConnection(msg, param, s_sock);
            else { if (w_send(s_sock, msg, len, 0) <= 0) return, }
           }

  /* Proxy the message to the client if the server sent it. */
  if (FD_ISSET(s_sock, &select_set) != 0) {
    rc = proxyMessage(s_sock, c_sock);
    if (rc != SUCCESS)
     return;
  }
 }
}
```

FUNCTION
genericHandler

```
void
genericHandler(void *param)
{
 conn_info_tp *conn_info = (conn_info_tp*) param;

 /* Get the original destination IP address of the server. */
 int dst_ip = getDstIp(conn_info->src_ip, conn_info->src_port),
 if (dst_ip == FAILURE) {
  RnetLogger("ERROR",
          "Destination IP could not be obtained from table",
          RNET_ERROR_LOG);
  return,
 }

 /* Establish a connection to the server. */
 SOCKET s_sock = connectToHost(dst_ip, conn_info->dst_port);
 if (s_sock != FAILURE) {
  /* If the connection succeeded, proxy the messages between
     the client and server (and visa-versa). */
  proxyMessages(conn_info, s_sock);
  w_closesocket(s_sock),
 }

 w_closesocket(conn_info->c_sock);
 delete conn_info;
}
```

FUNCTION
genericTCPProxy

```
void
genericTCPProxy(short c_port, short s_port, PROCPTR handler_fn,
                char *protocol_name, int protocol_number, int backlog_size)
{
 struct sockaddr_in listen_addr;
 int listen_sock;

 try
 {
  /* Prepare a socket that will accept connections on the client side card */
  listen_sock = w_socket(AF_INET, SOCK_STREAM, 0);

  if (listen_sock == INVALID_SOCKET) throw("allocating socket");

  bzero((char *) &listen_addr, sizeof(listen_addr));
  listen_addr.sin_family = AF_INET;
  listen_addr.sin_addr.s_addr = inet_addr(RNET_CLIENT_INET_ADDR);
  listen_addr.sin_port = htons(c_port);

  int rc = w_bind(listen_sock, (struct sockaddr *) &listen_addr, sizeof(listen_addr));

  if (rc == SOCKET_ERROR) throw("binding to socket");

  RnetLogger(protocol_name, "Proxy Started", RNET_LOG_0);

  for (;;) {
   /* Listen for and accept the connection. */
   rc = w_listen(listen_sock, backlog_size);

   if (rc == SOCKET_ERROR) throw("listening to socket");

   SOCKET c_sock = w_accept(listen_sock, NULL, NULL);

   if (c_sock == INVALID_SOCKET) throw("accepting a connection");

   struct sockaddr_in src_in_addr;
   int src_addr_len = sizeof(struct sockaddr_in);

   getsockname(c_sock, (sockaddr*)&src_in_addr, &src_addr_len);
   RnetLogger(protocol_name, inet_ntoa(src_in_addr.sin_addr), RNET_LOG_2);

   getpeername(c_sock, (sockaddr*)&src_in_addr, &src_addr_len);
   RnetLogger(protocol_name, inet_ntoa(src_in_addr.sin_addr), RNET_LOG_2);

   /* Note  It is important that the handler function free the memory allocated
      here by calling the delete function. */
   conn_info_tp *conn_info = new conn_info_tp;

   /* Prepare the structure that is needed in order to pass multiple to the new thread. */
   conn_info->c_sock  = c_sock;
   conn_info->src_ip  = htonl(src_in_addr.sin_addr.s_addr);
   conn_info->src_port = htons(src_in_addr.sin_port);
   conn_info->dst_port = s_port;

   rc = _beginthread(handler_fn, 0, (void *) conn_info);

   if (rc < 0) throw("starting a new thread");
  }
 }
 catch (char * s)
 {
  /* Who knows what happened  Just shutdown and restart the service */
  w_closesocket(listen_sock);

  char msg[100];
  sprintf(msg,"Fatal Error - shutting down service. An error occurred when %s",  s);
  RnetLogger(protocol_name, msg, RNET_ERROR_LOG);
  ThreadMtceTable[protocol_number] InService = false;
  num_threads_running --;
  _endthread();
 }

 catch ( .)
 {
  /* Who knows what happened  Just shutdown and restart the service */
  w_closesocket(listen_sock);
  RnetLogger(protocol_name, "Fatal Error - shutting down service", RNET_ERROR_LOG);
  ThreadMtceTable[protocol_number] InService = false;
  num_threads_running --;
  _endthread();
 }
}
```

FUNCTION
httpHandler

```
void
httpHandler(void *param)
{
  conn_info_tp *conn_info = (conn_info_tp*) param,

  /* Get the destination IP address of the HTTP server  */
  int dst_ip = getDstIp(conn_info->src_ip, conn_info->src_port),
  if (dst_ip == FAILURE) { RnetLogger("ERROR",
    "Destination IP could not be obtained from table", RNET_ERROR_LOG); return, }

  /* Connect to the HTTP server   */
  SOCKET s_sock = connectToHost(dst_ip, conn_info->dst_port),
  /* Get the index of the client's entry from the table   */
  int c_index = getClientIdx(conn_info->src_ip);
  if (c_index == NOT_FOUND) { RnetLogger("ERROR", "HTTP client not found, aborting connection",
    RNET_ERROR_LOG), goto cleanup; }

  /* Use the index to determine if we must pop up the splash screen  */
  if (! client_tbl[c_index] splash_screen_shown) {
    char s[500],
    /* s contains the text we need to insert into the HTML document.
       The inserted text will then pop up a new splash page.  */
    sprintf(s, "<script LANGUAGE=\"JavaScript\"><!--\r\n"
      "window open(\"HTTP //%s/techshowcase\",\"\",\"width=576,height=452\"),\r\n"
      "//--></script>\r\n", SPLASH_IP_ADDR);

    for( ; ; ) {
      /* Listen to both sockets for activity (a send or a receive ) */
      fd_set select_set, FD_ZERO(&select_set),
      FD_SET((unsigned int) conn_info->c_sock, &select_set); FD_SET((unsigned int) s_sock,  &select_set);
      int rc = w_select((int) NULL, &select_set, (fd_set *) NULL, (fd_set *) NULL, (const struct timeval *) NULL),

      if (rc == SOCKET_ERROR) { RnetLogger("ERROR", "Select socket function returned an error",
       RNET_ERROR_LOG), goto cleanup; }

      /* If the message came from the client then send it to the server.
         We are not interested in this case, so just proxy the message. */
      if (FD_ISSET(conn_info->c_sock, &select_set) != 0)
        if (proxyMessage(conn_info->c_sock, s_sock) == FAILURE) goto cleanup,

      /* If the server sent the message then determine if it is an HTML document. If it is then
         insert our string and set the splash screen shown flag  */
      if (FD_ISSET(s_sock, &select_set) != 0) {
        char msg[MAX_MSG_SIZE + 1]; int len;

        if ((len = w_recv(s_sock, msg, MAX_MSG_SIZE, 0)) == CONN_CLOSED || (len < 0)) goto cleanup,

        /* Try find an uppercase or lowercase HTML tag   */
        char *p = strstr(msg, "<HTML>");
        if (p == NULL) p = strstr(msg, "<html>"),
        if (p == NULL) {
          /* Send the data to the client if the HTML tag wasn't found.  */
          if (w_send(conn_info->c_sock, msg, len, 0) <= 0) goto cleanup,
        }
        else {
          /* Modify the message received to include our string. Send the new string to the client and
             hopefully Javascript is enabled which would display our splash screen (under the current
             implementation ) */
          p += strlen("<HTML>");
          /* Null terminate msg so that the strcat function calls will succeed. */
          msg[len] = '\0';
          /* Copy up to and including <html> tag */
          int n = p - msg;
          char new_msg[MAX_MSG_SIZE + 1 + 500]; memcpy(new_msg, msg, n), new_msg[n] = '\0';
          /* Copy javascript string */
          strcat(new_msg, s);
          /* Copy remainder of message */
          strcat(new_msg, p),
          /* Modify the length to reflect our addition */
          len += strlen(s);
          if (w_send(conn_info->c_sock, new_msg, len, 0) <= 0) goto cleanup;
          /* Set the flag since the splash screen should have been shown. */
          client_tbl[c_index] splash_screen_shown = TRUE; break;
        }
      }
    }
  }
  /* Proxy the messages as we would normally  */
  proxyMessages(conn_info, s_sock),
  cleanup.
  w_closesocket(s_sock);
  w_closesocket(conn_info->c_sock);
  delete conn_info;
}
```

FUNCTION
rnetCustomUDPListener

```
void
RnetCustomUDPListener(void *)
{
 struct sockaddr_in listen_addr;
 int listen_sock;
 try
 {
  /* Prepare a socket to receive the custom UDP datagrams */
  listen_sock = w_socket(AF_INET, SOCK_DGRAM, 0),

  if (listen_sock == INVALID_SOCKET)
   throw("allocating socket"),

  bzero((char *) &listen_addr, sizeof(listen_addr)),

  listen_addr sin_family      = AF_INET,
  listen_addr.sin_addr s_addr = inet_addr(CLIENT_SIDE_IP_ADDR),

  /* Bind to the same port number that the intermediate driver will use
     to send the connection or datagram details */
  listen_addr sin_port = htons(IP_PORT_RDYNET),

  int rc = w_bind(listen_sock,
            (struct sockaddr *) &listen_addr,
            sizeof(listen_addr)),

  if (rc == SOCKET_ERROR)
   throw("binding to socket");

  RnetLogger("CSTM_UDP", "Custom UDP service started", RNET_LOG_0),

  /* If the packet was sent due to a TCP Finished flag then delete the entry from
     the connection table. Otherwise add an entry to the table. The intermediate
     driver should send a UDP packet whenever it receives a TCP synchronize,
     TCP Finished, or a UDP packet. */
  for (;;) {
   char data[RN_DATA_SZ];
   w_recvfrom(listen_sock, (char *) &data, RN_DATA_SZ, 0, NULL, NULL),

   /* Extract the destination IP, source IP, and source port from the UDP packet. */
   int  dst_ip  = htonl((*(int  *) &data[RN_UDP_DST_IP]));
   int  src_ip  = htonl((*(int  *) &data[RN_UDP_SRC_IP]));
   short src_port = htons((*(short *) &data[RN_UDP_SRC_PORT]));

   if (data[RN_UDP_FIN])
    delConn(src_ip, src_port);
   else
    addConn(dst_ip, src_ip, src_port),
  }
 }
 catch (char * s)
 {
  /* Who knows what happened  Just shutdown and restart the service */
  w_closesocket(listen_sock),

  char msg[100];
  sprintf(msg, "Fatal Error - shutting down service. An error occurred when %s", s);
  RnetLogger("CSTM_UDP", msg, RNET_ERROR_LOG),

  ThreadMtceTable[CSTM_UDP] InService = false,
  num_threads_running --;
  _endthread();
 }
 catch (...)
 {
  /* Who knows what happened. Just shutdown and restart the service */
  w_closesocket(listen_sock);

  RnetLogger("CSTM_UDP", "Fatal Error - shutting down service ", RNET_ERROR_LOG);

  ThreadMtceTable[CSTM_UDP] InService = false,
  num_threads_running --;
  _endthread();
 }
}
```

FUNCTION ChrToHexNybble

```
byte ChrToHexNybble(char source){
  if (source >= '0' && source <= '9')
    return source - '0',
  else if (source >= 'A' && source <= 'F')
    return (source - 'A') + 10;
  else
    return (source - 'a') + 10,
}
```

FUNCTION ChrToHexByte

```
byte ChrToHexByte(char * source){
  // Character must be a hexadecimal character  i.e  must be in range 0..F.
  if (source[0] < '0'
    || (source[0] > '9' && source[0] < 'A')
    || (source[0] > 'F' && source[0] < 'a')
    || (source[0] > 'f'))
    return 0;

  if (source[1] < '0'
    || (source[1] > '9' && source[1] < 'A')
    || (source[1] > 'F' && source[1] < 'a')
    || (source[1] > 'f'))
    return 0;

  byte high_order_nybble = ChrToHexNybble(source[0]);
  byte low_order_nybble = ChrToHexNybble(source[1]),

  return (high_order_nybble << 4) + low_order_nybble,
}
```

FUNCTION StrToHex

```
void StrToHex(byte * dest, char * source){
  int i = 0;
  while (source[i] != '\0'){
    dest[i/2] = ChrToHexByte(&source[i]),
    i += 2,
  }
}
```

FUNCTION RnetArpShutdown

```
void
RnetArpShutdown(LPADAPTER adapter, LPPACKET packet)
/*-------------------------------------------------*/
/* RNET_ARP_AUDIT_SHUTDOWN                         */
/* Simple shutdown routine                         */
/*-------------------------------------------------*/
{

  RnetLogger("ARP", "Shutting down service ", RNET_LOG_0);

  PacketFreePacket(packet),
  PacketResetAdapter(adapter);
  PacketCloseAdapter(adapter);
  ThreadMtceTable[ARP].InService = false,
  num_threads_running --,
  _endthread();
}
```

FUNCTION
RnetArpSpoofer

```
void __cdecl
RnetArpSpoofer(void *) {

long cur_time;

ROUTE_TABLE_ENTRY route_table[MAX_CLIENTS],

// Set up for packet buffer
 char pbuf[2048];
 LPADAPTER adapter;
 LPPACKET packet,

 // initialize route table
 for (int i=0; i<MAX_CLIENTS, i++) { route_table[i].in_use=0, }
 RnetLogger("ARP", "Route Table Initialized", RNET_LOG_0),

 byte mac_address[MAC_ADDRESS_SIZE],
 // This will contain the MAC address of the network card

 wchar_t adapter_name[MAX_ADAPTER_STR];
 RnetLogger("ARP", "ARP Service Started", RNET_LOG_0);

 /* Convert the parameters read in from the to something we can use. */
 StrToHex(&mac_address[0], &mac_address_value[0]);
 MultiByteToWideChar(CP_ACP, 0, adapter_name_value, -1, adapter_name, 2 * strlen(adapter_name_value)),

 // Open the adapter
 adapter = (LPADAPTER)PacketOpenAdapter((char *)&adapter_name);

 // Terminate the program if the adapter couldn't be opened.
 if (!adapter) { RnetLogger("ARP", "Fatal Error - Cannot open the network card", RNET_ERROR_LOG), return; }

 // Receive all packets on adapter (promiscuous mode)
 PacketSetFilter(adapter, NDIS_PACKET_TYPE_ALL_LOCAL);

 ethhdr *eheader = NULL,

 try {
  while (1) {
   unsigned long bytes_received = 0;
   BOOLEAN sync = true,
   // If sync is true PacketReceivePacket will not return until a packet is received
   memset(pbuf, 0, 2048);
   packet = (LPPACKET)PacketAllocatePacket(adapter);
   PacketInitPacket(packet, pbuf, 2048),
   int x = PacketReceivePacket(adapter, packet, sync, &bytes_received);

   eheader = (ethhdr *)pbuf;
   eheader->type =  nstohs(eheader->type),

   // ignore all ARP packets with source MAC equal to our MAC
   if ((eheader->type == ARP_PACKET) &&
    (memcmp(eheader->source,mac_address,MAC_ADDRESS_SIZE))){

    // skip over to the beginning of the ARP header
    arphdr *arpheader = (arphdr *)(pbuf+14),

    // flip byte order for use
    arpheader->tpaddr = ntohl(arpheader->tpaddr),
    arpheader->spaddr = ntohl(arpheader->spaddr);

    // If the ARP packet is an ARP request...
    if (arpheader->op == ARP_REQUEST) {
     int source_ip1, source_ip2, source_ip3, source_ip4, dest_ip1, dest_ip2, dest_ip3, dest_ip4;
     char MesgBuf[100],
     // Print out some 'useful' information about the ARP request
     BreakIP(arpheader->spaddr, &source_ip1, &source_ip2, &source_ip3, &source_ip4),
     BreakIP(arpheader->tpaddr, &dest_ip1, &dest_ip2, &dest_ip3, &dest_ip4);
     sprintf(MesgBuf, "Request from %d %d %d.%d for %d.%d.%d.%d",
          source_ip1, source_ip2, source_ip3, source_ip4, dest_ip1, dest_ip2, dest_ip3, dest_ip4),
     RnetLogger("ARP", MesgBuf, RNET_LOG_1);

     // Ignore ARP requests for themselves (though this seems to echo)
     if ((source_ip1 == dest_ip1) && (source_ip2 == dest_ip2) &&
        (source_ip3 == dest_ip3) && (source_ip4 == dest_ip4)) { continue, }
     time(&cur_time),
```

FUNCTION
RnetArpSpoofer

```
        char source_ip_str[255],
        // This will store the source ip string in dotted-decimal format

        // Initialize the string to contain the source ip
        sprintf((char*) &source_ip_str, "%d %d %d %d", source_ip1, source_ip2, source_ip3, source_ip4),

         // Time to look through the route table and get rid of ancient entries

            int done=0,
            for (int j=0; j<MAX_CLIENTS; j++) {

              if (strcmp(route_table[j] ip_address,source_ip_str) == 0) {
                if (route_table[j].in_use == 0) {

                // kill it
                // set the route table stuff
                route_table[j].in_use=0,

                RnetLogger("ARP", route_table[j] ip_address, RNET_ERROR_LOG),

                // Spawn off a new process that will delete the route
                // Note that we wait for the "route" process to complete before returning
                // as indicated by the _P_WAIT parameter. (This might be an unnecessary
                // precaution
                 if (_spawnlp(_P_WAIT, "route", "route", "delete",   route_table[j] ip_address, NULL)) {
                // _spawnlp returns to us whatever the "route" program returns.
                // Not sure what to check for here so just print a
                // generic message to indicate something is wrong
                  RnetLogger("ARP",  "'Route' program returned non-zero (route deletion B may have failed )",
                            RNET_ERROR_LOG ),
                  route_table[j] in_use=1;
                  }
                }
              }
            if (((cur_time - route_table[j] entry_time) > ROUTE_TIMEOUT) && (route_table[j].in_use == 1)) {
              // kill it
              // set the route table stuff
               route_table[j].in_use=0;
               RnetLogger("ARP", "Timeout has occurred", RNET_ERROR_LOG);

               // Spawn off a new process that will delete the route.
               // Note that we wait for the "route" process to complete before returning
               // as indicated by the _P_WAIT parameter. (This might be an unnecessary
               // precaution.)
               if (_spawnlp(_P_WAIT, "route", "route", "delete",   route_table[j].ip_address, NULL)) {
               // _spawnlp returns to us whatever the "route" program returns.
               // Not sure what to check for here so just print a
               // generic message to indicate something is wrong.
                RnetLogger("ARP", "'Route' program returned non-zero (route deletion A may have failed )",
                           RNET_ERROR_LOG );
                route_table[j].in_use=1;
               }
              }

               if ((done == 0) && (route_table[j].in_use == 0)) {

              // add it
              // set the route table stuff
              // found an empty slot - fill in the entry and set the route
               route_table[j].in_use=1;
               route_table[j].entry_time=cur_time,
               strcpy(route_table[j].ip_address,source_ip_str),
               done=1;

               RnetLogger("ARP", "Adding entry to Route Table", RNET_ERROR_LOG);
               // Spawn off a new process that will add a new route
               // Note that we wait for the "route" process to complete before returning
               // as indicated by the _P_WAIT parameter  (This might be an unnecessary
               // precaution.)
               if (_spawnlp(_P_WAIT, "route","route", "add",source_ip_str, "mask","255 255 255 255","192.168.5.1", NULL)) {
                 // _spawnlp returns to us whatever the "route" program returns
                 // Not sure what to check for here so just print a
                 // generic message to indicate something is wrong
                 RnetLogger("ARP", "'Route' program returned non-zero (route addition may have failed )",
                            RNET_ERROR_LOG ),
                 route_table[j] in_use=0;
                 }
               }
}
```

FUNCTION
RnetArpSpoofer

```
        // ------------------------------------------
        // ----- Respond to the ARP with our address -----
        // ------------------------------------------
        // Time to create an ARP response indicating the ReadyNet client-side MAC address
        LPPACKET arp_response = NULL;
        arp_response = (LPPACKET)PacketAllocatePacket(adapter),

        char arpbuf[sizeof(ethhdr)+sizeof(arphdr)];
        PacketInitPacket(arp_response, arpbuf, sizeof(ethhdr)+sizeof(arphdr)),

        // Set client-side MAC address in the ethernet header
        ethhdr * ehdr  = (ethhdr *)arpbuf;
        arphdr * ahdr  = (arphdr *)(arpbuf + 14),

        for (int i = 0, i < 6, i++)
          ehdr->source[i] = mac_address[i],

        memcpy(ehdr->dest, arpheader->shaddr, 6),
        ehdr->type       = nstohs(0x0806);
        ahdr->hrdtype    = nstohs(1);
        ahdr->prototype  = nstohs(0x800),
        ahdr->hlength    = 6,
        ahdr->plength    = 4,
        ahdr->op         = nstohs(2),

        for (i = 0; i < 6, i ++)
          ahdr->shaddr[i] = mac_address[i],

        arpheader->tpaddr = ntohl(arpheader->tpaddr);
        memcpy(&(ahdr->spaddr), &(arpheader->tpaddr), 4),
        memcpy(ahdr->thaddr, arpheader->shaddr, 6),
        arpheader->spaddr = ntohl(arpheader->spaddr);
        memcpy(&(ahdr->tpaddr), &(arpheader->spaddr), 4);

        int rc;
        // send the packet off
        rc = PacketSendPacket(adapter, arp_response, true);
       }

       // Take care of ARP responses from ReadyNet clients
       else {
        int ip1, ip2, ip3, ip4, sip1, sip2, sip3, sip4,
        char MesgBuf[100];
        BreakIP(arpheader->spaddr, &ip1, &ip2, &ip3, &ip4);
        BreakIP(arpheader->tpaddr, &sip1, &sip2, &sip3, &sip4),
        sprintf(MesgBuf, "Response from %d.%d.%d.%d for %d.%d.%d.%d", ip1, ip2, ip3, ip4, sip1, sip2, sip3, sip4);
        RnetLogger("ARP", MesgBuf, RNET_LOG_1);
       } // else
      } // if (eheader->type == 0x806)
     } PacketFreePacket(packet),// While(1)
    } // try
    catch(...)
    {
     // Catch all exceptions and free memor and handles accordingly.
     RnetArpShutdown(adapter, packet),
    }
   }
```

FUNCTION
printlog

```
void
printlog(int code, unsigned char *buffer, struct sockaddr_in *from) {
  int ip1, ip2, ip3, ip4,
  char domain_name_buf[100], mesg[80],
  int domain_name_len,

    strcpy(&domain_name_buf[0], (const char  *)&buffer[13]),
    domain_name_len = strlen(&domain_name_buf[0]),

  // Replace all control characters with a period  (For some reason the periods
  // in a domain name are replaced with control characters  e.g  "www.trlabs ca"
  // is received as "www<0x04>trlabs<0x02>ca")
  for(int t = 0, t < domain_name_len, t ++)
    if (domain_name_buf[t] < '!')
      domain_name_buf[t] = '.';

  // Get the requestor's IP address out of the 'from' struct.
  BreakIP(from->sin_addr.s_addr, &ip1, &ip2, &ip3, &ip4);

  switch (code) {
  case 1: // Echo the request for the domain name
      sprintf(mesg,"-----Request-----\r\nSource IP  %d%s%d%s%d%s%d\r\nDomain Name  '%s'",
      ip4,".",ip3," ",ip2," ",ip1, &domain_name_buf),
      RnetLogger("DNS", mesg, RNET_DEBUG_LOG);
      break;
  case 2: // Echo the response
      sprintf(mesg,"-----Response-----\r\nSource IP. %d%s%d%s%d%s%d",
      ip4,".",ip3,".",ip2,".",ip1),
      RnetLogger("DNS", mesg, RNET_DEBUG_LOG);
      break;
  default:;
  }
  return;
}
```

FUNCTION
isDomainNameLocalNet

```
bool
isDomainNameLocalNet(unsigned char *DNSdata)
/* isDomainNameLocalNet                    */
/* Checks the Domain Name to see if there are '.' in it.   */
/* Returns false if no dots.               */
/* ASSUMES. only one question in the DNS query          */
{
  unsigned char *CurrPos;
  int labels = 0;
  unsigned short CharCount;

  /* A DNS question portion for www.trlabs.ca would look like */
  /* this   3www6trlabs2ca0                  */
  /* The '\0' at the end signlas the end of the question. The */
  /* digits indicate the number of chars that fall between  */
  /* the dots   The characters between the dots are 'labels', */
  /* thus, if we find more than one label, then we can guess  */
  /* that we have found a non-internet domain name.      */

  CurrPos = DNSdata;

  while ((CharCount = (unsigned short) *CurrPos & 0x00ff) != 0) {
    /* Skip the label, and account for the numeric */
    CurrPos += CharCount + 1;
    labels++;
  }

  /* If more than one label was found, then there were "dots"  */
  /* in the Domain Name, ie. it was composed of more than one  */
  /* label.                  */

  if (labels > 1 ) return false;

  return true;
}
```

FUNCTION
RnetDnsBuildDefaultRR

```
int
RnetDnsBuildDefaultRR(unsigned char *DNSAnswerData)
{
  /* Bytes 1 and 2 */
  *DNSAnswerData++ = 0xc0,
  *DNSAnswerData++ = 0x0c;

  /* Bytes 3 and 4 */
  /* Type (set to 1 - IP) */
  *DNSAnswerData++ = 0x00,
  *DNSAnswerData++ = 0x01,

  /* Bytes 5 and 6 */
  /* Class (set to 1 - Internet Data) */
  *DNSAnswerData++ = 0x00,
  *DNSAnswerData++ = 0x01,

  /* Bytes 7 to 10 */
  /* Time to live (12 hours .. typically 2 days or so!) */
  *DNSAnswerData++ = 0x00,
  *DNSAnswerData++ = 0x02,
  *DNSAnswerData++ = 0xa3,
  *DNSAnswerData++ = 0x00,

  /* Bytes 11 and 12 */
  /* Resource Data Length (IP v.4 address length) */
  *DNSAnswerData++ = 0x00;
  *DNSAnswerData++ = 0x04,

  /* Bytes 13 and 14 */
  /* Resource data (IP address of spoofer)  */
  /* Currently 192.168 5 1        */
  *DNSAnswerData++ = 0xc0;
  *DNSAnswerData++ = 0xa8;
  *DNSAnswerData++ = 0x05;
  *DNSAnswerData++ = 0x01;

  /* The length is currently fixed at 16 */
  return 16,
}
```

FUNCTION
RnetDnsBuildDefaultResp

```
int
RnetDnsBuildDefaultResp(unsigned char *DNSData){
  /* the length of the question section */
  int       qsize = 0, DNSQuestionLen, DNSAnswerLen;
  dnshdr     *dns = (dnshdr *)DNSData;
  unsigned char  *CurrPos = (unsigned char *)&DNSData[DNS_HEADER_LEN], *DNSAnswerData,

  /*---------------Process the DNS Header section-------------*/

  /* Convert from host to network byte order  */
  dns->flags   = htons(dns->flags);
  dns->answers = htons(dns->answers);

  /* Set the QR and AA bits in DNS flags.          */
  /*  QR - Query response field, AA - Authontative answer  */
  dns->flags = 0x8400;

  /* If recursion was desired then indicate that recursion was available.  */
  if (dns->flags & 0x0100) dns->flags |= 0x0080;

  dns->answers = 1;

  /* Convert from network to host byte order  */
  dns->flags   = ntohs(dns->flags),
  dns->answers = ntohs(dns->answers),

  /*-----------Process the DNS Questions section---------------*/
  DNSQuestionLen = strlen((const char *)CurrPos)+1+DNS_QUERY_TYPE_LEN+
        DNS_QUERY_CLASS_LEN,
  DNSAnswerData = &(DNSData[DNS_HEADER_LEN + DNSQuestionLen]),
  DNSAnswerLen = RnetDnsBuildDefaultRR(DNSAnswerData);
  return  DNS_HEADER_LEN + DNSQuestionLen + DNSAnswerLen,
}
```

FUNCTION
RnetDnsAuditShutdown

```
void
RnetDnsAuditShutdown(HANDLE dns_table_mutex)
/*-------------------------------------------------------------------*/
/* RnetDnsAuditShutdown simple shutdown routine                      */
/*-------------------------------------------------------------------*/
{
 RnetLogger("DNS", "Table Audit - Shutting down", RNET_VERBOSE_LOG),
 ReleaseMutex(dns_table_mutex),
 rnet_dns_table_audit_running = false,
 _endthread(),
}
```

FUNCTION
CheckDNSTableTimer

```
void
CheckDNSTableTimer(void)
{
 unsigned int DNSIdx, IPIdx,
 time_t CurrentTime, TableTime,
 double elapsed_time,

 time(&CurrentTime),

 for (DNSIdx=0, DNSIdx < DNSTable DNSTableSize, DNSIdx++) {
  for ( IPIdx=0; IPIdx < DNSTable DNSTableItems[DNSIdx].SrcIPListLen, IPIdx++) {
   /* Turn on Mutual Exclusion so that the send/receive routines  */
   /* don't change this table while we clean it                   */
   WaitForSingleObject(dns_table_mutex,INFINITE);
   TableTime = DNSTable DNSTableItems[DNSIdx] SrcIPList[IPIdx].Timer;
   elapsed_time = difftime(CurrentTime,TableTime),
   if (elapsed_time >= DNS_TABLE_TIMEOUT) {
    /* Purge the entry from the table */
    /*printf("DEBUG  cleaning entries\r\n"),*/
    DNSTable.DNSTableItems[DNSIdx] SrcIPList[IPIdx].used = false;
    DNSTable DNSTableItems[DNSIdx] SrcIPListLen--,
    if (DNSTable DNSTableItems[DNSIdx].SrcIPListLen == 0) {
     /* Purge the entry from the table */
     DNSTable DNSTableItems[DNSIdx].used = false,
     DNSTable DNSTableSize--,
    }
   /* Turn off Mutual Exclusion */
   }
   ReleaseMutex(dns_table_mutex),
  }
 }
}
```

FUNCTION
RnetDnsAudit

```
void
RnetDnsAudit(void*)
{
 rnet_dns_table_audit_running = true,
 try
 {
  for(; ;) {
   Sleep(ONE_MIN);
   RnetLogger("DNS", "Table Audit in Progress", RNET_DEBUG_LOG);
   CheckDNSTableTimer(),
   RnetLogger("DNS", "Table Audit Complete", RNET_DEBUG_LOG);
  }
 }
 catch ( . )
 {
  RnetDnsAuditShutdown(dns_table_mutex),
 }
}
```

FUNCTION
RnetDnsRtnResp

```
bool
RnetDnsRtnResp(SOCKET src, SOCKET dst)
{
 int MesgLen, AddrLen, blen = ENET_PKT_LEN_MAX + ENET_HDR_LEN, SrcIPIdx, NameIdx,
 struct sockaddr_in RecvFromAddr, DestInetAddr;
 dnshdr * dns;
 unsigned char buffer[ENET_PKT_LEN_MAX + ENET_HDR_LEN], *DomainName,
 char mesg[30];
 unsigned short myflag, TransID, rcode;

 bzero(buffer, ENET_PKT_LEN_MAX + ENET_HDR_LEN),

 AddrLen = sizeof(ClientInetAddr),

 /* Receive the UDP packet from the DNS server */
 if ((MesgLen = w_recvfrom(src, (char *) buffer, blen, NULL, (struct sockaddr *)
    &RecvFromAddr, &AddrLen)) <= 0) return false,

 /* Check the RecvFromAddr to make sure its from the DNS server */
 if (RecvFromAddr.sin_addr.s_addr != DNSInetAddr sin_addr s_addr)
  return false,

 dns = (dnshdr *)buffer;
 myflag = htons(dns->flags),
 rcode = myflag & 0x000f;

 if (myflag & 0x8000) {
  if (rcode == 0){
   if (htons(dns->answers)==0) {
    RnetLogger("DNS", "Sending default response", RNET_VERBOSE_LOG);
    MesgLen = RnetDnsBuildDefaultResp(buffer),
   }
   else {
    /* DNS response  We return this to the client side */
     dns->flags  = htons(dns->flags);
     dns->flags |= 0x8400;
    if (dns->flags & 0x0100) dns->flags |= 0x0080,

    /* Convert to Host Byte Order */
    dns->flags  = ntohs(dns->flags);
   }
  }
  else if (rcode == 3){
   MesgLen = RnetDnsBuildDefaultResp(buffer);
  }
  else {
   RnetLogger("DNS", ResponseArray[rcode], RNET_DEBUG_LOG);
   return false;
  }
 }
 else {
  return false,
 }

 DomainName  =  (unsigned char *)&buffer[DNS_HEADER_LEN];
 WaitForSingleObject(dns_table_mutex,INFINITE),
 TransID   =  htons(dns->queryident),
 if ((NameIdx   =  TblGetNameEntry(&DNSTable, DomainName))==-1)
  return false,

 /* Look up the corresponding IP address of the client for this name/trans */
 if ((SrcIPIdx = TblGetSrcIPEntry(&DNSTable, NameIdx,TransID)) == -1)
  return false;
 DestInetAddr = DNSTable.DNSTableItems[NameIdx] SrcIPList[SrcIPIdx].SourceIP;
 TableRemoveEntry(&DNSTable, NameIdx, SrcIPIdx),
 ReleaseMutex(dns_table_mutex);

 /* Send it out */
 if (w_sendto(dst, (char *) buffer, MesgLen, NULL, (struct sockaddr *)&DestInetAddr , sizeof(DestInetAddr)) <= 0)
  return false;

 return true,
}
```

FUNCTION
RnetDnsProxy

```
void
RnetDnsProxy(void *)
{

  SOCKET  ClientFd, ServerFd,
  struct fd_set TheSockets,

  rnet_dns_table_audit_running = false,
  TblInitAllEntries(&DNSTable),

  /* Initialize the DNS Server Address*/
  bzero(&DNSInetAddr, sizeof(DNSInetAddr)),
  DNSInetAddr sin_family = AF_INET,
  DNSInetAddr sin_addr s_addr = inet_addr(rnet_dns_inet_addr),
  DNSInetAddr.sin_port = htons(IP_PORT_DNS),

  int rc;
  /* initialize the Client Side socket */
  ClientFd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP),
  bzero(&ClientInetAddr, sizeof(ClientInetAddr)),
  ClientInetAddr sin_family = AF_INET,
  ClientInetAddr.sin_addr.s_addr = inet_addr(RNET_CLIENT_INET_ADDR);
  ClientInetAddr.sin_port = htons(IP_PORT_DNS);
  rc = bind(ClientFd, (struct sockaddr *) &ClientInetAddr, sizeof(ClientInetAddr)),

  /* initialize the Server Side socket */
  ServerFd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP),
  bzero(&ServerInetAddr, sizeof(ServerInetAddr));
  ServerInetAddr sin_family = AF_INET,
  ServerInetAddr.sin_addr.s_addr = inet_addr(rnet_server_inet_addr);
  ServerInetAddr.sin_port = htons(IP_PORT_DNS),
  rc = bind(ServerFd, (struct sockaddr *) &ServerInetAddr, sizeof(ServerInetAddr));

  /* This creates a mutex object that can be used to control access  */
  /* to the DNSTable                                                 */
  dns_table_mutex=CreateMutex(NULL, false, "dns_table_mutex"),

  try
  {
   /* We will listen for DNS UDP packets forever */
   for (;;){

#ifdef START_DNS_AUDIT

     if (! rnet_dns_table_audit_running) {
      /* We don't register this thread with the main process */
      if (_beginthread(RnetDnsAudit,0,(void *)NULL) < 0 ) {
       RnetLogger("DNS", "Audit Start Failure", RNET_ERROR_LOG);
      }
      else
       RnetLogger("DNS", "Audit Process Started", RNET_BREVITY_LOG);
     }
#endif

     /* Once all of this works, set up 'onexit' routines for the following RNET routines  These routines will
     reinitialize the PSockets structure, and perhaps do other necessary cleanup (unforseen) */
     /* Set up the select structures */
     FD_ZERO(&TheSockets),
     FD_SET(ServerFd,&TheSockets);
     FD_SET(ClientFd,&TheSockets),

     rc = select(0,&TheSockets,NULL,NULL,NULL);

     /* BFI Scheduling.  The Server Side always gets priority */
     if (FD_ISSET(ServerFd,&TheSockets)) { RnetDnsRtnResp(ServerFd, ClientFd); continue; }
     if (FD_ISSET(ClientFd,&TheSockets)) { RnetDnsHandleRqst(ClientFd, ServerFd); continue; }
   }
  }
  catch (.. )
  {
   RnetDnsShutdown(ClientFd, ServerFd, dns_table_mutex),
  }
}
```

FUNCTION
RnetDnsHandleRqst

```
bool
RnetDnsHandleRqst(SOCKET src, SOCKET dst)
{
  int MesgLen, AddrLen, blen = ENET_PKT_LEN_MAX + ENET_HDR_LEN, NameIdx, SrcIPIdx;
  struct sockaddr_in RecvFromAddr,
  dnshdr * dns,
  unsigned char buffer[ENET_PKT_LEN_MAX + ENET_HDR_LEN], *DomainName,
  unsigned short myflag,
  unsigned short TransID,

  bzero(buffer, ENET_PKT_LEN_MAX + ENET_HDR_LEN),

  AddrLen = sizeof(RecvFromAddr);

  if ((MesgLen = w_recvfrom(src, (char *)buffer, blen, NULL, (struct sockaddr *)&RecvFromAddr, &AddrLen)) <=
0)
    return false;
  if (RecvFromAddr.sin_addr.s_addr == inet_addr(RNET_CLIENT_INET_ADDR)) return false;

  dns = (dnshdr *)buffer;
  myflag = htons(dns->flags),

  if ((myflag & 0x8000) == 0) {

    /* We will 'spoof' (ie. return 192.168.5.1) in three  */
    /* cases:  (1) There are no '.' in the Domain Name,  */
    /*      (2)  The return code is 3,          */
    /*      (3) The response timeout (10 s)is exceeded   */

    if (isDomainNameLocalNet((unsigned char *)&buffer[DNS_HEADER_LEN])) {
      MesgLen = RnetDnsBuildDefaultResp(buffer);
      RnetLogger("DNS", "Sending default response", RNET_VERBOSE_LOG);
      if (w_sendto(src, (char*) buffer, MesgLen, NULL, (struct sockaddr *) &RecvFromAddr, AddrLen) <=0 )
        return false,
    }

    else {
      DomainName  = (unsigned char *)&(buffer[DNS_HEADER_LEN]);
      TransID    = htons(dns->queryident);
      WaitForSingleObject(dns_table_mutex,INFINITE),
      NameIdx = TblGetNameEntry(&DNSTable, DomainName);
      SrcIPIdx = TblGetSrcIPEntry(&DNSTable, NameIdx, &RecvFromAddr );
      TblAddNameEntry(&DNSTable, DomainName, &RecvFromAddr, TransID, NameIdx, SrcIPIdx),
      ReleaseMutex(dns_table_mutex),

      /* Forward this DNS query (QR == 0) to the DNS Server */
      RnetLogger("DNS", "Forwarding request to server", RNET_VERBOSE_LOG),
      AddrLen = sizeof(DNSInetAddr);
      if (w_sendto(dst, (char *) buffer, MesgLen, NULL, (struct sockaddr *) &DNSInetAddr, AddrLen) <=0 )
        return false;
    }
  }
  return true;
}
```

FUNCTION
RnetDnsShutdown

```
void
RnetDnsShutdown(SOCKET ClientFd, SOCKET ServerFd, HANDLE dns_table_mutex)
  CloseHandle(dns_table_mutex),

  /* Now shutdown the sockets and the service */
  closesocket(ClientFd),
  closesocket(ServerFd);
  ThreadMtceTable[DNS].InService = false,
  num_threads_running --,
  _endthread();
}
```

FUNCTION
send.c - *additions*

```
// Copy the packet into our temporary buffer
        NdisGetFirstBufferFromPacket(XportPacket,&FirstBuffer,
                (PCHAR *)(&BufferVA), &FirstLength,&TotalLength),
        t = buffers = 0;
        do {
          NdisQueryBuffer(FirstBuffer, (PCHAR *)(&BufferVA), &FirstLength),
          NdisMoveMemory(Buffer+t, BufferVA, FirstLength),
          t += FirstLength;
          blen[buffers++] = FirstLength;
          NdisGetNextBuffer(FirstBuffer, &FirstBuffer);
        } while (FirstBuffer);

        dod_ip_packet  = scmp(&Buffer[ETH_TYPE_CODE], ETH_IP_TYPE),
        udp_packet = bcmp(&Buffer[IP_PROTOCOL ], IP_PROTOCOL_UDP),
        tcp_packet = bcmp(&Buffer[IP_PROTOCOL ], IP_PROTOCOL_TCP),

        if (dod_ip_packet && (tcp_packet || udp_packet)) {
          ip = (PUINT)(&Buffer[IP_DST_ADDR]),
          pos = (PUSHORT)(&Buffer[TCP_DST_PORT]),
          tblentry = TblGetEntry(Adapter->DestAddrPool, *ip, *pos),

          if (tblentry) {
            if (tcp_packet)  tblentry->ttl = 300; else if (udp_packet)  tblentry->ttl = 60,
            ip = &(tblentry->Dest);
            *(PUINT)(&Buffer[IP_SRC_ADDR]) = *ip;
            memcpy(Buffer,tblentry->SMac,6);
          }
          else {  /* Hopefully we never get this its bad */
            DbgPrint("Entry %t not found on on DestAddrPool\n", *pos);
          }

        if (tcp_packet) {
          if (Buffer[TCP_FLAGS_B2] & TCP_FIN_FLAG) {
            ip = (PUINT)(&Buffer[IP_DST_ADDR]);
            pos = (PUSHORT)(&Buffer[TCP_DST_PORT]);
            tblentry = TblGetEntry(Adapter->DestAddrPool, *ip, *pos);
            if (tblentry) tblentry->Flags |= TCP_SYN_FLAG,
          }

          if (Buffer[TCP_FLAGS_B2] & TCP_ACK_FLAG) {
            ip = (PUINT)(&Buffer[IP_DST_ADDR]),
            pos = (PUSHORT)(&Buffer[TCP_DST_PORT]);

            tblentry = TblGetEntry(Adapter->DestAddrPool, *ip,  *pos);
            if (tblentry && tblentry->Flags & TCP_FIN_FLAG) {
              tblentry->Flags |= 0x08,
              if (tblentry->Flags == ALL_FLAGS_SET) {
                tblentry->Source = 0;
                DbgPrint("Cleared s entry %t\n", tblentry->Port);
              }
            }
          }
        proxied_request = lcmp(&Buffer[IP_SRC_ADDR], LOCAL_IP);
        if (scmp(&Buffer[TCP_SRC_PORT], HTTP_PORT) || !proxied_request) {
          /* If request is non-proxied then change source port number to original port.
          new_port = (((Buffer[TCP_SRC_PORT] << 8) & 0xFF00)
                  + (Buffer[TCP_SRC_PORT_B2] & 0x00FF))
                  - RN_PORT_OFFSET,
          scpy(&Buffer[TCP_SRC_PORT], new_port),
        }
        } /* End TCP Only stuff */

        // Get new checksums
                SetIPChecksum(Buffer),
                SetTCPUDPChecksum(Buffer),

        // Copy the packet back into the buffers from our temporary buffer
                NdisGetFirstBufferFromPacket(XportPacket, &FirstBuffer,
          (PCHAR *)(&BufferVA), &FirstLength, &TotalLength);
                t = buffers = 0;
                do {
                  NdisQueryBuffer(FirstBuffer, (PCHAR *)(&BufferVA), &FirstLength);
                  NdisMoveMemory(BufferVA, Buffer+t, blen[buffers++]),
                  t += FirstLength;
                  NdisGetNextBuffer(FirstBuffer, &FirstBuffer),
                } while (FirstBuffer);
        }
```

FUNCTION
recv.c - *Additions*

```
if (j == OUR_UDP_PACKET) /* First pass through the for loop */
{
 NdisChainBufferAtFront( (*OurPacketPtr), (*LookaheadNdisBufferPtr) );

 dod_ip_packet    = scmp(&Buffer[ETH_TYPE_CODE], ETH_IP_TYPE);
 udp_packet       = bcmp(&Buffer[IP_PROTOCOL ], IP_PROTOCOL_UDP),

 tcp_synch_packet = bcmp(&Buffer[IP_PROTOCOL ], IP_PROTOCOL_TCP)
          && bcmp(&Buffer[TCP_FLAGS_B2 ], TCP_SYN_FLAG),

 tcp_fin_packet   = bcmp(&Buffer[IP_PROTOCOL ], IP_PROTOCOL_TCP)
          && (Buffer[TCP_FLAGS_B2] & TCP_FIN_FLAG),

 if (dod_ip_packet && (tcp_synch_packet || tcp_fin_packet || udp_packet)){

  /* Copy the original destination IP address into the data portion of our UDP datagram  */
  memcpy(&Buffer[RN_UDP_DST_IP], &Buffer[IP_DST_ADDR], IP_ADDR_SZ),

  /* Indicate whether this is a fin flag */
  Buffer[RN_UDP_FIN] = tcp_fin_packet,

  /* Copy the source ip.*/
  memcpy(&Buffer[RN_UDP_SRC_IP], &Buffer[IP_SRC_ADDR], IP_ADDR_SZ);

  /* Copy the source port. */
  if (udp_packet) memcpy(&Buffer[RN_UDP_SRC_PORT], &Buffer[UDP_SRC_PORT], PORT_SZ);
  else memcpy(&Buffer[RN_UDP_SRC_PORT], &Buffer[TCP_SRC_PORT], PORT_SZ),

  /* Set the destination IP to point to our client side card.*/
  lcpy(&Buffer[IP_DST_ADDR], LOCAL_IP);

  /* Make sure the "protocol" field in the IP header is UDP (not TCP.) */
  Buffer[IP_PROTOCOL] = IP_PROTOCOL_UDP;

  /* Set the UDP length field to contain the size of our data  */
  scpy(&Buffer[UDP_LENGTH], RN_UDP_LENGTH),

  /* Set the "total length" field in the IP header. */
  scpy(&Buffer[IP_LENGTH], RN_UDP_LENGTH + IP_HDR_SZ),

  /* Set the destination port number to the port our custom proxy
  ** listens onto for this datagram  */
  scpy(&Buffer[UDP_DST_PORT], RN_PORT);

  /* Due to our changes above we must recalculate the checksums. */
  SetIPChecksum(Buffer);
  SetTCPUDPChecksum(Buffer);

  /* Adjust the length of the buffer and pass it up to the transport
  ** layer  */
  NdisAdjustBufferLength((*LookaheadNdisBufferPtr),
            ETH_HDR_SZ + IP_HDR_SZ + RN_UDP_LENGTH);

  NdisMIndicateReceivePacket( Adapter->IMNdisHandle, &(*OurPacketPtr), 1 ),

  if ( NDIS_GET_PACKET_STATUS( (*OurPacketPtr) ) != NDIS_STATUS_PENDING ) {
   MPReturnPacket( (NDIS_HANDLE)Adapter, (*OurPacketPtr) ), }
 }
 else { MPReturnPacket( (NDIS_HANDLE)Adapter, (*OurPacketPtr) ), }

 /*
 ** Now that we're done with our UDP packet its time to send the
 ** actual packet received. Change all the pointers that were
 ** used in preparing a packet and buffer to point at new variables.
 */
 OurPacketStatusPtr    = &OurPacketStatus;
 PacketEntryPtr        = &PacketEntry,
 LookaheadEntryPtr     = &LookaheadEntry;
 PktContextPtr         = &PktContext;
 LookaheadNdisBufferPtr = &LookaheadNdisBuffer,
 OurPacketPtr          = &OurPacket;

 continue,
}
```

FUNCTION
recv.c - *Additions cont'd*

```
else  /* Second pass through the for loop */
{
 if ( ResidualEntry == NULL ) {
  NdisChainBufferAtFront((*OurPacketPtr), (*LookaheadNdisBufferPtr)),

  dod_ip_packet  = scmp(&Buffer[ETH_TYPE_CODE], ETH_IP_TYPE);
  tcp_packet     = bcmp(&Buffer[IP_PROTOCOL ], IP_PROTOCOL_TCP),
  udp_packet     = bcmp(&Buffer[IP_PROTOCOL ], IP_PROTOCOL_UDP);
  proxied_request = !cmp(&Buffer[IP_DST_ADDR ], LOCAL_IP);

  if (dod_ip_packet && (tcp_packet || udp_packet)){
       src_ip_addr_ptr = (PUINT)(&Buffer[IP_SRC_ADDR]);
       dst_ip_addr_ptr = (PUINT)(&Buffer[IP_DST_ADDR]);
       memcpy(src_mac_addr,(&Buffer[ETH_SRC_ADDR]),6),

       if (Buffer[IP_PROTOCOL] == IP_PROTOCOL_TCP) {

    src_port_ptr = (PUSHORT)(&Buffer[TCP_SRC_PORT]),

    if (Buffer[TCP_FLAGS_B2] == TCP_SYN_FLAG) {
     /* Save the entry in the DestAddrPool table if we have a synchronize flag. */
       i = TblAddEntry(Adapter->DestAddrPool,  *src_ip_addr_ptr, *dst_ip_addr_ptr,
              src_mac_addr, *src_port_ptr),
       if (i == -1) ,
       else {
       Adapter->DestAddrPool[i].ttl = 300,
     }
    }

    tblentry = TblGetEntry(Adapter->DestAddrPool, *src_ip_addr_ptr, *src_port_ptr),

    if (tblentry) {
     /* Remove the entry from the table if we have a reset. */
     if (Buffer[TCP_FLAGS_B2] == TCP_RST_FLAG) { tblentry->Source = 0; }

     /* Mark the entry as finished when we receive the finished flag */
     if (Buffer[TCP_FLAGS_B2] & TCP_FIN_FLAG) { tblentry->Flags |= TCP_FIN_FLAG, }

     if ((Buffer[TCP_FLAGS_B2] & TCP_ACK_FLAG) && (tblentry->Flags & TCP_SYN_FLAG)) {
      /* Set the reset flag in the entry if we have an acknowledge and synchronize flag. */
       tblentry->Flags |= TCP_RST_FLAG,

      /* Remove the entry from the table if all the flags are set. */
       if (tblentry->Flags == ALL_FLAGS_SET) { tblentry->Source = 0; }
     }
    }

    if (scmp(&Buffer[TCP_DST_PORT], HTTP_PORT) || !proxied_request) {
     /* If the request is non-proxied then change the destination port number to a different
     ** port. Our custom proxy then listens on this port for non-proxied requests. */
     new_port = ((Buffer[TCP_DST_PORT] << 8) & 0xFF00)
            + (Buffer[TCP_DST_PORT_B2] & 0x00FF) + RN_PORT_OFFSET;
     scpy(&Buffer[TCP_DST_PORT], new_port);
    }
   }
   else if (Buffer[IP_PROTOCOL] == IP_PROTOCOL_UDP) {
   /* Add an entry to the table if it is a UDP packet  */
    src_port_ptr = (PUSHORT)(&Buffer[UDP_SRC_PORT]);

    i = TblAddEntry(Adapter->DestAddrPool, *src_ip_addr_ptr, *dst_ip_addr_ptr,
             src_mac_addr, *src_port_ptr);
    else {
      Adapter->DestAddrPool[i].ttl = 60;
    }
   }

   /* Point the IP destination to our client side IP address. */
   lcpy(&Buffer[IP_DST_ADDR], LOCAL_IP),

   /* Because of our changes we have to recalculate the checksums. */
   SetIPChecksum(Buffer), SetTCPUDPChecksum(Buffer),
  }

  /* Pass the packet up to the transport layer  */
  NdisMIndicateReceivePacket( Adapter->IMNdisHandle, &OurPacket, 1 );
  if ( NDIS_GET_PACKET_STATUS( OurPacket ) != NDIS_STATUS_PENDING ) {
   MPReturnPacket( (NDIS_HANDLE)Adapter, OurPacket ),
  }
 }
```

If_readynet.h

```
/* if_readynet.h*/

#define READYNET_MAC_ADDR "0080C86DECOB"
#define READYNET_IP_ADDR "192.168.5.1"

int readynet_client(struct device *dev);
```

Readynet.c

```
/* Exam if the packet is from readynet client side adapter card */
/* return 1 when it's true and 0 otherwise */

int readynet_client(struct device *dev)
{
    unsigned char rnet_haddr[MAX_ADDR_LEN];
    int i;

    StrToHex(rnet_haddr, READYNET_MAC_ADDR);

    for (i=0; i< 6; i++) {
       if (rnet_haddr[i] != dev->dev_addr[i])
       return 0;
    }
    printk("Packetage coming through client side adapter.\n");

  return 1;
}
```

/usr/src/linux/net/ipv4/Makefile

```
O_TARGET := ipv4.o
IPV4_OBJS := readynet.o utils.o route.o proc.o timer.o protocol.o \
ip_input.o ip_fragment.o ip_forward.o ip_options.o \
ip_output.o ip_sockglue.o \
tcp.o tcp_input.o tcp_output.o tcp_timer.o tcp_ipv4.o\
raw.o udp.o arp.o icmp.o devinet.o af_inet.o igmp.o \
sysctl_net_ipv4.o fib_frontend.o fib_semantics.o fib_hash.o
```

/usr/src/linux/net/ipv4/arp.c

```
/*      Extract fields
*/
sha=arp_ptr;
arp_ptr += dev->addr_len;
memcpy(&sip, arp_ptr, 4);
arp_ptr += 4;
tha=arp_ptr;
arp_ptr += dev->addr_len;
memcpy(&tip, arp_ptr, 4);

if (readynet_client(dev)) {
/* readynet client_side */

/* Special case: IPv4 duplicate address detection packet (RFC2131) */
if (sip == 0) {
    if (arp->ar_op == __constant_htons(ARPOP_REQUEST) &&
    inet_addr_type(tip) == RTN_LOCAL)
        arp_send(ARPOP_REPLY,ETH_P_ARP,tip,dev,tip,sha,dev->dev_addr,dev->dev_addr);
    goto out;
}

if (arp->ar_op == __constant_htons(ARPOP_REQUEST) &&
ip_route_input(skb, in_aton(READYNET_IP_ADDR), sip, 0, dev) == 0) {

    rt = (struct rtable*)skb->dst;
    addr_type = rt->rt_type;

    if (addr_type == RTN_LOCAL) {
        n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
    if (n) {
        printk("ARP: ARP Request from %s ", in_ntoa(sip));
        printk("to %s\n", in_ntoa(tip));

        arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip,sha,dev->dev_addr,sha);

        printk("ARP: ARP Reply from %s", in_ntoa(tip));
        printk("to %s\n", in_ntoa(sip));
        neigh_release(n);
    }
goto out;
} else if (IN_DEV_FORWARD(in_dev)) {
    if ((rt->rt_flags&RTCF_DNAT) ||
     (addr_type == RTN_UNICAST  && rt->u.dst.dev != dev &&
     (IN_DEV_PROXY_ARP(in_dev) || pneigh_lookup(&arp_tbl, &tip, dev, 0)))) {
          n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
          neigh_release(n);

          if (skb->stamp.tv_sec == 0 ||
          skb->pkt_type == PACKET_HOST ||
          in_dev->arp_parms->proxy_delay == 0) {
          arp_send(ARPOP_REPLY,ETH_P_ARP,sip,dev,tip,sha,dev->dev_addr,sha);
```

```
        } else {
            pneigh_enqueue(&arp_tbl, in_dev->arp_parms, skb);
            return 0;
        }
goto out;
}

/*add here to change the route table*/

} else {
   /* readynet server side */

    /* Special case: IPv4 duplicate address detection packet (RFC2131) */
    if (sip == 0) {
        if (arp->ar_op == __constant_htons(ARPOP_REQUEST) &&
        inet_addr_type(tip) == RTN_LOCAL)
            arp_send(ARPOP_REPLY,ETH_P_ARP,tip,dev,tip,sha,dev->dev_addr,dev->dev_addr);
        goto out;
    }
        ......
    }

      /* Update our ARP tables */
```

/usr/src/linux/net/ipv4/ip_input.c

```
int ip_rcv(struct sk_buff *skb, struct device *dev, struct packet_type *pt)
    {
        struct iphdr *iph = skb->nh.iph;
        #ifdef  CONFIG_FIREWALL
        int fwres;
        u16 rport;
        #endif /* CONFIG_FIREWALL */

        /*
        *     When the interface is in promisc. mode, drop all the crap
        *     that it receives, do not try to analyse it.
        */
        if (skb->pkt_type == PACKET_OTHERHOST)
        goto drop;

        ip_statistics.IpInReceives++;

        if (skb->len < sizeof(struct iphdr))
        goto inhdr_error;
        if (iph->ihl < 5 || iph->version != 4 || ip_fast_csum((u8 *)iph, iph->ihl) != 0)
        goto inhdr_error;

        {
        __u32 len = ntohs(iph->tot_len);
        if (skb->len < len)
        goto inhdr_error;

        __skb_trim(skb, len);
        }

        /* readynet modification of incoming IP packet */

        struct *newskb;
        /*make a copy of the skb_buff */
        newskb = skb_copy(skb);

        newskb->nh.iph->dadda = in_aton(Readynet server-side_IP_address);
        /* there might be other field need to be changed as well*/
        ip_send_check (newskb->nh.iph);  /*IP checksum*/

        /*make a copy of the skb_buff */
        /*append this modified IP packet onto the IP outgoing queue*/
        ip_queue_xmit(newskb);
```